McGill University
School of Computer Science
Sable Research Group

Bytecode Testing Framework for SableVM Code-copying Engine

Sable Technical Report No. 2007-9

Gregory B. Prokopski, Etienne M. Gagnon, Christian Arcand
Sable Research Group
School of Computer Science, McGill University
Montreal, Quebec, Canada

November 16, 2007

Abstract

Code-copying is a very attractive, simple-JIT, highly portable fast bytecode execution technique with implementation costs close to a classical interpreter. The idea is to reuse, at Virtual Machine (VM) runtime, chunks of VM binary code, copy them, concatenate and execute together at a greater speed. Unfortunatelly code-copying makes anwarranted assumptions about the code generated by the compiler used to compile the VM. It is therefore necessary to find which VM code chunks can be safely used after being copied and which can not. Previously used manual testing and assembly analysis were highly ineffective and error prone. In this work we present a test suite, Bytecode Testing Framework (BTF) that contains a comprehensive set of fine-grained Java assembly tests. These tests are then used in conjunction with SableVM JVM testing mode to semi-automatically detect one-by-one which code chunks are safe to use. With this techunique we ported SableVM's code-copying engine to several architectures and had external contributors with no VM knowledge port SableVM to new architectures within as little as one hour. The presented Bytecode Testing Framework greatly improved reliability and ease of porting of SableVM's code-copying engine, thus its practical usability.

1 Introduction

Testing is a necessary part of any software development. The more advanced or complicated a system is the more testing it tends to require. Depending on the type of a system and current stage of development testing can be done by developers themselves, by designated specialists (also programmers), or, eventually, by end users. Testing can be done either in an ad-hoc manner, where tester tests system functionality case-by-case with little or no support from pre-existing tools, or with support of these. Use of testing support tools favors creation of sets of tests that can be executed automatically and report on findings.

In this work[1] we are concerned with testing of a particular element of a larger system—*copy-copying engine* as a part of Java Virtual Machine interpreter, SableVM [6]. Code-copying has been proposed as a VM interpreter implementation technique that improves performance, reducing the low costs vs. high performance gap between interpreters and compilers [5, 9]. Depending on an application and other factors the *code-copying*[2] technique can give from 1.15 to 2.14 times speedup [6] over the *direct-threading* technique. While the code-copying itself has its roots in interpreters it actually involves dynamic creation of code, therefore it can be viewed as a simple Just-In-Time compiler.

Unfortunately code-copying, comes with a serious issue. It makes optimistic and unwarranted assumptions about the code generated by compiler used to compile the JVM. The core idea of code-copying is to reuse, at VM runtime, chunks of VM binary code corresponding to Java bytecodes. These *code chunks* are copied into a newly allocated place in memory, concatenated and executed together to achieve superior performance while keeping the design very simple and largely architecture-agnostic, thus ensuring high portability of the VM. The problem arises because the C standard[3] does not contain any semantics that would allow us to express and impose necessary restrictions on selected parts of code. The labels placed before and after the code chunks and used as start and end pointers during code copying do not guarantee contiguity of the resulting binary code chunks, nor do they place restrictions on the use of relative addressing. These and other closely related issues become more and more important as compilers use more aggressive optimizations. Code chunks that are not functionally equivalent after being copied to a new location in memory can not be used by code-copying engine and have to be executed in a slower manner, without being copied.

[1] The work described in this document was done in years 2002-2004, yet, for various reasons, it was not published at that time.

[2] Note that in the literature what we call code-copying is sometimes referred to as *inlining* or *inline-threading* [5]; these latter terms, however, we find, suggest method or function inlining to most compiler developers and researchers.

[3] SableVM, as many system tools, is written in C.

It is therefore necessary to identify these problematic code chunks and bytecode instructions they correspond to so that the rest of code chunks could be used to achieve higher speed with code-copying. Initially the implementations of code-copying we know of were tested manually, by trial-and-error, or by carefully looking at the assembly of VM binary code, or used hand-written, unportable assembly. These methods do not give satisfying results in terms of the knowledge required to use them, their time efficiency, resistance to human error, general usability, and portability. For example on each different architecture every compiler version can potentially compile code in a different manner, resulting in a different set of code chunks that can be copied. With fully manual testing approach it is not feasible to test a VM with every compiler version change. SableVM, for example, uses internally over 300 bytecodes, which means over 300 code chunks that can be potentially used by code-copying, but need to be deemed safe-to-copy before they are used. The goal of this work is to give the testing results higher trust index and introduce as much automatization into the testing process as possible.

With this work we make the following contributions:

- we identify problems arising from the use of highly optimizing compilers that undermine the assumptions necessary for code-copying engines to function,
- we present the design of a custom testing suite targeted at finding JVM code that does not hold the properties necessary for code used in code-copying,
- we present a semi-automatic method of testing code chunks that involves mixed-mode execution that dynamically alternates between safe, simpler and slower direct-threading and faster copy-copying.

The rest of this work is structured as follows. In the next section we describe the related work. In Section 3 we describe in more detail types of execution engines and give a broader view of the problematic issues. In Section 4 we present the design of our Bytecode Testing Framework followed by experimental results in Section 6. We close with conclusions and future work description in Sections 7 and 8.

2 Related work

In our work we are concerned with ensuring proper operation of VMs using the code-copying technique. This technique originates from the *direct-threaded* interpretation and was first described by Piumarta and Riccardi in their work on, what they called, *selective inlining* [9]. Compilers used at that time did not use too many aggressive optimizations that would make code-copying impossible, therefore testing was not of such importance.

Gagnon was the first to use the code-copying technique in a Java interpreter [5, 6]. This implementation solved some important problems specific to the interpretation of Java bytecode. As can be seen in Figure 1, it also included three execution engines: simplest, plain C *switch-based*, direct-copying and code-copying (a.k.a. *inline-threaded*). Interestingly, experiments done with a simple, non-optimizing portable JIT for SableVM (SableJIT [1]) showed that such a JIT was only barely able to achieve speeds comparable to the code-copying engine. This demonstrated once again that code-copying is a very attractive solution performance-wise.

One of the important reasons why code-copying is significantly faster than other interpretation techniques is its positive influence on the success rate of branch predictors commonly used in today's hardware containing branch target buffers (BTB). Ertl showed in his work on indirect branch prediction in interpreters [2, 4]

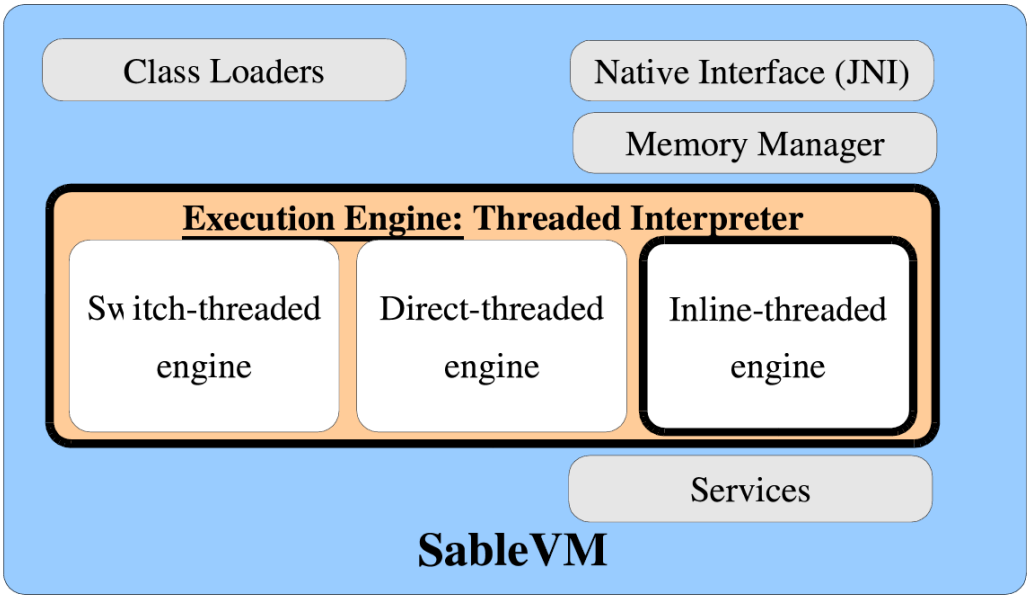

Figure 1: SableVM is a complete Java virtual machine featuring three execution engines: from most portable switch-threaded to fastest code-copying (also known as *inline-threaded*).

that other solutions improving branch prediction, e.g. bytecode duplication, can also give significant performance improvement. Speedup due to branch prediction improvements much outweighs other negative effects such as increased instruction-cache misses.

A solution similar to code-copying engine is a JIT using code generated by a C compiler developed by Ertl [3]. In this solution, however, the pieces of code were actually modified (patched) on the fly, so as to contain immediate values and remove the need for the instruction counter. Due to the patching architecture-specific code was necessary. Interestingly Ertl's solution did include automated tests to detect code chunks that were definitely not usable for code-copying, but it was not guaranteed to find all such chunks.

A brief discussion of the initial work done within SableVM JVM on ensuring usability of its code-copying engine was presented by Gagnon [7]. He touched on the main issues regarding conflicts between goals of highly-optimizing compilers and the requirements for code used by code-copying engines.

There exist several Java-related testing suits. Mauve, a subproject of GNU Classpath, groups tests targeted mainly at testing of equivalence of its class libraries with proprietary Java libraries. In this field the most comprehensive testing is offered by Java Compatibility Kit available from Sun. Kaffe JVM also has its own set of library tests, and more interestingly a subset of tests aiming to ensure its JIT compiler is working properly. Also GCJ (part of GNU Compiler Collection) has its own test suite. All these test suits target specifically either Java libraries compatibility or general VM compiler saneness. As will become evident later, they are all unsuitable for the fine-grained approach necessary to test a code-copying engine.

Another class of tests are various Java benchmarks, like SPEC JVM98, CaffeineMark, SciMark 2 and many others created mainly as tools for performance testing. While useful in general, these benchmarks are of little help while testing a code-copying engine.

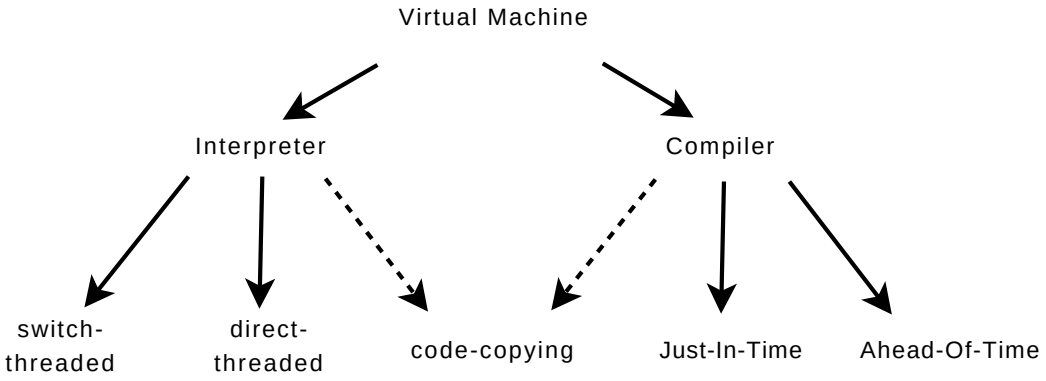

Figure 2: The taxonomy of Virtual Machines execution engines.

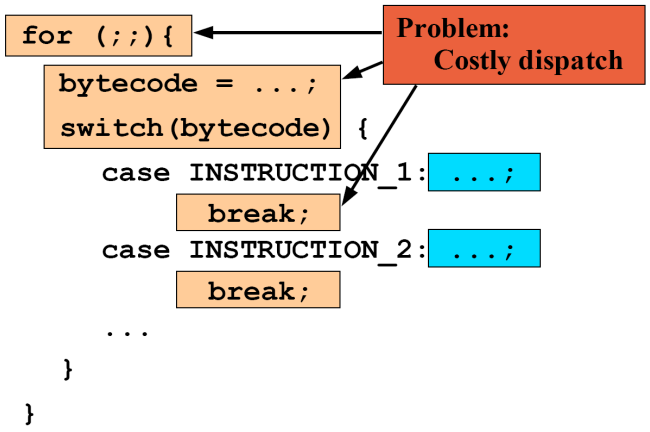

Figure 3: Plain-C, switch-threaded interpreter incurs large runtime overhead.

3 Background

Interpreters have the advantage of simplicity, although improved performance is possible with different design approaches. We illustrate the main designs on the left side of Figure 2 to situate the code-copying approach; these include a basic *switch-threaded* interpreter, and a *direct-threaded* model.

A *switch-threaded* interpreter simulates a basic fetch, decode, execute cycle, reading the next bytecode to execute and using a large *switch-case* statement to branch to the actual VM code appropriate for that bytecode. This process is straightforward but if, such as in Java, bytecodes often encode only small operations the overhead of fetching and decoding an instruction is proportionally high, making the overall design quite inefficient, as shown in Figure 3.

A *direct-threaded* interpreter is a more advanced interpreter that minimizes decoding overhead. This kind of interpreter requires an extension offered by some compilers known as *labels-as-values*. Normally a C program can contain *goto*s only to labels. With the labels-as-values extension it is possible to take an address of a label and store it in a variable. Later this variable can be used as an argument of a *computed goto*. In a direct-threaded interpreter a stream of bytecodes is thus replaced by a stream of addresses of

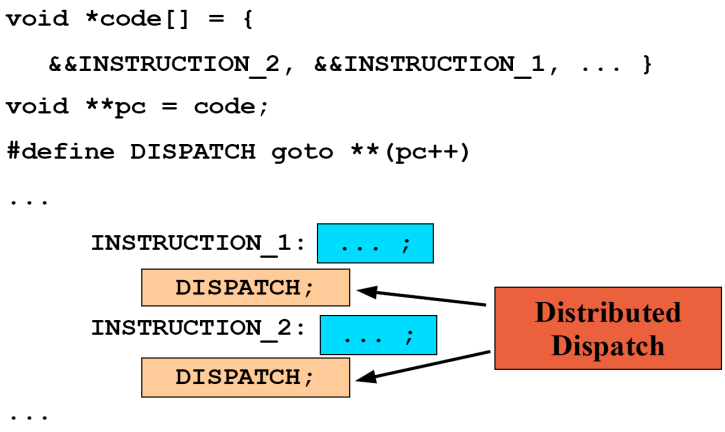

Figure 4: Direct-threaded interpreter translates stream of bytecodes into stream of addresses before execution and lowers the repetitive interpretation overhead.

labels. The labels themselves are placed at the start of code responsible for execution of operations encoded by each bytecode. With this mechanism an interpreter can immediately execute a *computed goto* jumping directly to a chunk of code of the next instruction, as illustrated in Figure 4. Optimization is implied by reducing the repeated decoding of instructions, trading repeated test-and-branch sequences for a one-time preparatory action where a stream of bytecodes is translated into a stream of addresses.

In some sense, and as indicated in Figure 2, code-copying bridges interpreter and compiler-based VM implementation approaches. Code-copying is a further optimization to interpreter design, but one which makes relatively strong assumptions about compiler code generation. The basic idea of code-copying is to make use of the compiler applied to the VM to generate binary code for matching bytecodes. Parts or *chunks* of the VM code are used to implement the behavior of each bytecode. Those chunks of code are marked with labels at their begin and end. At runtime, the interpreter copies the binary chunks corresponding to an input stream of bytecodes and concatenates them into a new place in memory, as shown in Figure 5. Such a chain of concatenated instructions is called a superinstruction and it can execute at a much greater speed than using any of the other two formerly described techniques. Depending on an application and other factors the code-copying technique can give from 1.15 to 2.14 times speedup [6] over the direct-threaded technique.

As numerous studies have shown the performance gains from using code-copying technique are clear [2, 4–6, 9]. However one of the biggest problems the implementators of code-copying interpreter engines face is ensuring that the fragments of the code chunks copied to construct superinstructions are still fully functional in their new locations and as a part of a superinstruction. In particular, to behave correctly a code chunk must not contain relative jumps or calls to targets that would be outside of the chunk, and its control flow must start at the *top* and exit at the *bottom*. Chunks which do not possess these properties are not safe to copy and execute. Unfortunately, the C standard does not contain any semantics that would allow us to express and impose such restrictions on selected parts of code, therefore we need to resort to testing the code outputted by the compiler.

Before the testing can happen, however, it is necessary to clearly identify what are the common problems that need to be found by tests, how they manifest, and how they can be dealt with.

- *Basic blocks partitioning*. Optimizing compilers, like GCC 3.2 and newer, divide basic blocks into likely executed ones (hot) and not likely executed (cold). Blocks belonging to each group are put

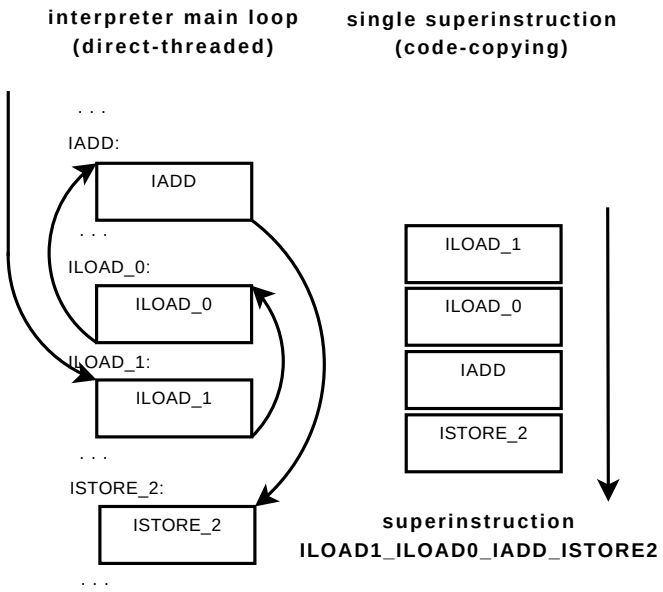

Figure 5: A simplified comparison of direct-threaded and code-copying engines.

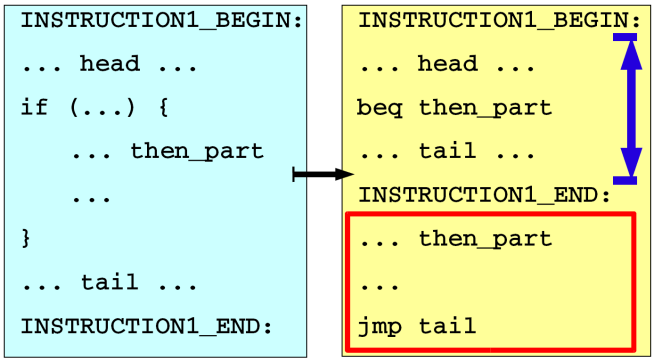

Figure 6: Optimizing compiler can relocate less likely executed code to the outside of labels bracketing code used by code-copying.

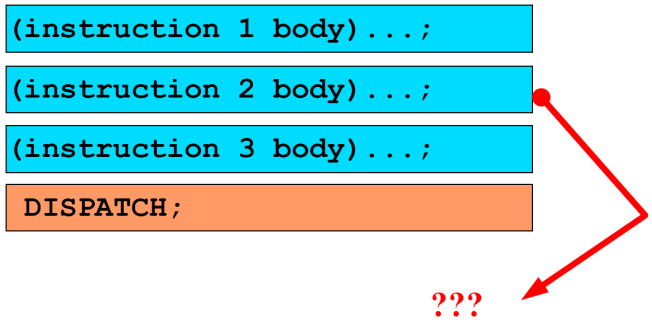

Figure 7: Execution of a superinstruction containing a code chunk with missing part or a call using relative addressing might cause VM crash.

together, so as to improve cache efficiency. Unfortunately this optimization often moves a basic block belonging to the internal control flow of a code chunk to the outside (usually far away) from of the bracketing labels of the code chunk thus making it unusable for code copying (see Figure 7). This might reduce the number of bytecodes usable for code-copying to almost zero. GCC 3.3 introduced an option to disable this optimization which is now used by SableVM by default to make code-copying of a reasonable number of bytecodes possible again.

- *Most often executed path optimization*. As illustrated in Figure 6 an optimizing compiler can relocate code that is less likely to be executed, like null pointer checks (common in many bytecodes) to the outside of pair of labels bracketing the code chunk. If this happens, such code chunk can not be used for code-copying. This is because the only code that is copied is the code between the two bracketing labels. When such code chunk is used (see Figure 7) in code-copying and the less likely execution path is encountered then the relocated part of code is missing from superinstruction an undefined behavior will occur resulting most likely a segmentation fault.

  SableVM features a special technique of trapping signals instead of explicit null pointer checks to remove this problem in some cases. It is possible to allow VM to cause a segmentation fault and then recover from it, which can serve as a costly way of handling an exceptional, thus rare, situation where an explicit null pointer check would be used. By removing the check and its associated conditional we removed the possibility that a compiler would relocate the less likely executed block (e.g. a null pointer check) to the outside of the labels bracketing the code chunk 6. The removal of the rarely needed check also has a generally positive effect on the performance **??**.

- *Jumps using relative addressing*. A regular C `goto` to a label can be translated by a compiler into an instruction using a relative or absolute addressing. If a relative addressing method is used then such bytecode is not suitable for code-copying, as the target of the jump is dependant on the position of the code, and this position is changed when the code is copied. To force an absolute jump SableVM forces the compiler to use a *computed goto*, which is part of the labels-as-values extension. This kind of goto takes a pointer variable as its parameter and executes an absolute jump to the specified target. Note that this technique can and should only be used for Goths whose targets are outside of a code chunk, which (in a Java VM) are mostly jumps to a signal handler. SableVM does make use of this technique.

- *Calls using relative addressing*. On many popular architectures, e.g. on Intel, the target address of a call is specified using address relative to the currently executed instruction. A code chunk containing such call can not be used for code-copying for the same reasons as in case of a relative jump. It is possible to change the call in a C program to use a call-by-value construct which also forces compiler to use an absolute address. Unfortunately not all calls are visible in the source code. On some architectures some math operations are performed via function calls, and not in place. Also on our set of benchmarks we noted no measurable performance improvement from making these problematic bytecodes usable for code-copying. SableVM, in its current version, makes no use of this workaround.

With all the above workarounds SableVM can only increase the likelihood of an instruction being usable for code-copying. The testing is still necessary to ensure to find out which instructions exactly can be used.

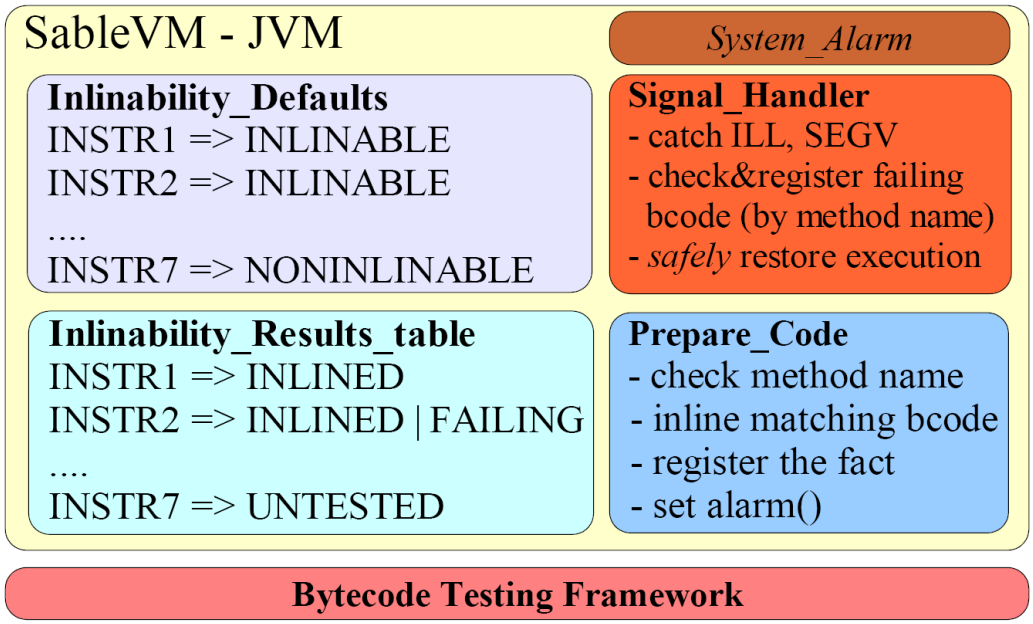

Figure 8: The architecture of Bytecode Testing Framework and Inlinability Testing Mode in SableVM

4 Bytecode Testing Framework design

The Bytecode Testing Framework (BTF) for SableVM has been created to ensure that none of the disasters described in the previous section happen. To that end we need to test every code chunk to ensure it will work properly when copied in every possible situation, that is—on its every control flow path. For every bytecode we are interested in using for code-copying (and thus every corresponding code chunk) we analyze the source code and find all control flow paths it contains. This is necessary to be able to create a series of tests that will exercise every control flow path of each bytecode.

4.1 Issues with bytecode testing

There are three important issues with this approach. First, in the testing suite we need to have complete control over what bytecodes are executed. We need to be sure that a method containing a test actually does contain a certain bytecode (the one we want to test). A javac compiler has often much too much freedom in choosing and optimizing Java code to give us the necessary control. Therefore we decided to write the tests directly in Java assembly using Jasmin [8] as the Java assembler tool.

The second issue is that because initially it is not known which code chunks are safe to be copied and which are not, then the JVM must be able to run without copying any code chunks. This is important, because to execute even the simplest method a JVM needs to bootstrap first. The bootstrap process, according to our measurements, means execution of several hundreds thousands of bytecodes. We modified SableVM to include a special compilation option called *bytecode testing mode*, part of which is a mixed-mode execution ability. When compiled with this option the code-copying is turned off for all bytecodes by default, and all bytecodes are interpreted using direct-threading. Copying of the code is turned on highly selectively, bytecode-at-a-time, and only for a single bytecode inside of each of the handcoded assembly test methods

of our test suite.

The last thing we need to keep in mind is that once the testing is complete the results need to be easily usable during compilation of normal version of SableVM with code-copying engine. Because the results vary depending on the compiler version and machine architecture we expect to end up with an extensive database containing information pertaining to each bytecode. Because of the large number of internally used bytecodes (several hundreds) and the support for multiple architectures we need to ensure a database design that is practically usable.

4.2 Special bytecode testing mode and testing suite

We modified SableVM to support mixed-mode execution where it is possible to decide at runtime whether to execute a bytecode using code-copying or direct-threading. When the test suite is executed SableVM recognizes special names of classes and methods containing the tests. In particular we made each test method name contains the name of a single bytecode. This bytecode is the only one of the whole method that will be executed using code-copying. This way execution of a method gives a clear answer on whether a particular bytecode was executed properly or not.

In the testing mode SableVM holds a runtime database of bytecodes along with flags describing the status of each bytecode:

- *not copyable* - do not even attempt to use it with code-copying, it is already known not to work,
- *copyable* - do attempt to use it with code-copying but it is not guaranteed to work, needs to be tested,
- *failing* - means a code chunk of this bytecode has been used with code-copying and failed one or more tests,
- *untested* - needs to be tested, but has not been yet.

  - initially set for all *copyable* bytecodes, cleared when the code chunk of this bytecode is used for code-copying, so that at the end of testing we can find bytecodes that were not tested, for example because test suite did not contain any tests for them.

Initially all bytecodes have either *not copyable* or *copyable* and *untested* flags set. The *untested* flag is cleared just before the execution of a test pertaining this bytecode is attempted. Bytecodes executed successfully by code-copying engine will have their *copyable* flag set and *failing*, *untested* flags cleared. Failed bytecodes will have their *failing* flag set and *untested* flag cleared.

Failure detection

Detection of improper execution of bytecodes uses two mechanisms. First, for each test method executed the expected result is known and compared against the result returned. If a method returns a different result then a test failure is reported. Second, more important detection method is meant to register code-copying execution failures, such as segmentation faults, illegal instructions, and infinite loops. Segmentation faults and illegal instruction execution attempt are detected by trapping UNIX OS signals. The infinite loops are detected by setting a UNIX system alarm. The alarm is reset on every method entry and when not reset, e.g. due to VM being stuck in an infinite loop, it times out also causing a signal. Together these two mechanisms are prepared to detect all possible execution failures, whether coming from code-copying or other problems.

Failure registration

When a failure occurs the signal handler looks up the name of a method being executed and from it derives the name of the failing bytecode and sets its *failing* flag.

Failure recovery

After the failure is registered the VM attempts to restore the execution of bytecode by creating and throwing a Java exception. This way, after the return from the signal handlers, the interpreter pops the Java stack frame and thus makes the interpretation return to the caller method, which manages the test execution. The latter method is prepared to handle the Java exception. To avoid interferences with the existing standard Java exceptions system we use own, non-standard exception to signal a code-copying failure. In the extreme cases recovery is not possible and the VM crashes when returning from the signal handler. To ensure that it is still possible to extract test results about the bytecodes tested up to the moment of VM crash we modified SableVM to produce debug output informing about failures as soon as each failure is identified.

4.2.1 Database of results

The usability of a code-chunk for code-copying engine is affected by the underlying platform, compiler version and selected options. SableVM uses internally over 300 bytecodes, some of which we never expect to be used for code-copying but at least half of these need to be tested and the results need to be stored. Because of that the database of information about bytecodes that can or can not be used for code copying for various version of compilers and various architectures was expected to be too big to be directly embedded into the JVM sources. The information stored in the database is optimized for human-readability and use. Each row in the database contains information about one bytecode, each column corresponds to a single architecture-and-compiler-version-and-options setup. We developed a set of `M4` preprocessor macros to transform the information stored in the database into C language. These macros produce one `.h` file per database column containing lines of `#define` constructs, one per bytecode.

5 Practical usage

This subsection is a concise how-to describing step by step the use of Bytecode Testing Framework and SableVM's Testing Mode.

5.1 Preparation

1. Open the inlinability.list file (the database) in an editor that supports "horizontal split" so that you can see two parts of the file at the same time.
2. Make the upper view few lines short and scroll the content to see the name of the architecture-compiler being tested. Scroll the lower view down until the architecture-compiler name matches with one of the column containing the data about bytecodes.
3. Clean up the column from old information by overwriting the data with spaces. Do not ever remove the `","` characters, also keep them in one column with other lines.

4. Save the file but do not close the close the editor as it will be useful later.

Important: When switching between using `--enable-inlinability-testing` and not using it it is *required* to execute `make distclean`. Otherwise execution errors will appear that make no sense at all and which might waste hours to debug. This is due to the fact that we change the way some `c` files are generated by `M4` preprocessor depending on this option, in particular whether a call to `no_inlining_increment_pc()` is generated into the source code or not.

5.2 Testing

1. Compile and install SableVM with these options:
   `--enable-inlinability-testing`
   `--with-threading=inlined`
   `--enable-signals-for-exceptions`.
2. Run Bytecode Testing Framework on SableVM.
3. The end results will clearly indicate which bytecodes are failing. Put `" NOT,"` in the column of your architecture for each such bytecode, then save the file. An empty entry `" ,"` indicates that a bytecode can be used for code-copying.
4. Compile SableVM omitting the `--enable-inlinability-testing` option (or changing it to `--disable-inlinability-testing`) and re-run Bytecode Testing Framework again to ensure the tests caught all the problems. If not—repeat the previous step until reaching the fixed-point.
5. Follow the "5.2 Testing" instructions once more but this time compile SableVM using `--disable-signals-for-exceptions`. Also this time put `" SIG,"` into the database if a bytecode fails without `signals-for-exception` but not with them.

5.3 Final steps

When the testing is finished and SableVM works properly using code-copying we ask you to publish your inlinability.list. Please do not use `diff` to show changes in the content, publish a complete file. Optionally you can modify configure.ac and make your architecture use code-copying by default:

```
case \${}host in
   alpha*-gnu)     with\_threading=inlined ;;
```

5.4 Troubleshooting

Be aware that whether a chunk of code can be used by code-copying depends on many factors, mainly on the architecture, the compiler, and its version. For each new compiler version the testing should be repeated to ensure VM robustness.

- During the tests SableVM has to deal with execution of random code and/or segmentation faults. These are not reliable conditions for execution, thus it might severely crash during tests and you

won't see the final table with results. As a remedy use `grep REGISTERING` on the output to get the list of the bytecodes that are failing before the crash and use that information to partially update the inlinability.list file. Recompile SableVM and repeat the testing procedure. There are high chances that your testing will progress further this time. Repeat until VM execution finishes properly with the final table of results.

- If you experience failures or exceptions while testing on an instruction that is either already marked as not for use by code-copying or is in the list of the instructions that are expected to never be usable by code-copying (at the bottom of inlinability.list) then it indicates a problem likely unrelated to code-copying. Recompile SableVM with one of the other execution engines and run BTF again.
- If with each run a substantial number (20-40 or more) of bytecodes fail (usually randomly) it probably means that the data/instruction cache flush function in SableVM is not working properly for your architecture.
- If SableVM has not been ported to your architecture at all then you should first ensure proper functioning of switch and direct-threaded engines.
- If your class library lacks `java/lang/InliningException.class` you will see `sablevm: cannot create vm` error. Note that this error might occasionally happen for other reasons.
- If you compile with GCC older than 3.3 you might need to use the `--disable-no-reorder-blocks` option, especially with GCC 2.95. Be warned that GCC 3.2 on some platforms generate many codes not usable for code-copying while not providing options introduced in GCC 3.3 that help alleviate the problem.
- Bytecode Testing Framework is also a regular testing suite. On some architectures it will for example detect problems with finite and infinite divisions. These are most likely not relate to code-copying and can be fixed while using switch or direct engine.
- When `signals-for-exceptions` are disabled SableVM might receive a signal not from bytecode execution failing due to code-copying but because of some unusual behavior of hardware. A good example is Intel's hardware that responds with a floating point error interrupt when execution the division: `Integer.MIN_VALUE / -1` . A bytecode executing such operation will be misleadingly listed as *failing*. When executed without the testing mode and without `signals-for-exceptions` a "Floating point exception" will be reported. We need to be aware of such behavior because it is possible that other architectures might have their own special cases.

6 Experimental results

The main reason for creation of Bytecode Testing Framework was to improve the reliability of code-copying engine and improve time efficiency of porting VMs using this kind of engine to new architectures and compilers. Formerly the reliability testing and porting were done manually, which often meant days of trial-and-error, sometimes including analysis of assembly code to understand and be able to avoid certain issues.

Since the creation of BTF the SableVM project has seen external contributors (C programmers with no VM knowledge) porting code-copying engine of SableVM to a completely new architecture within as little as

```
IASTORE ----------------------=> INLINED and WORKING (RECognizable)
LASTORE ----------------------=> INLINED and WORKING (RECognizable)
FASTORE ----------------------=> INLINED and WORKING (RECognizable)
DASTORE ----------------------=> INLINED and WORKING (RECognizable)
AASTORE ----------------------=> FAILING (RECognizable)
BASTORE ----------------------=> UNTESTED - NONILINABLE (NOT recognizable)
CASTORE ----------------------=> INLINED and WORKING (RECognizable)
SASTORE ----------------------=> INLINED and WORKING (RECognizable)
IADD ------------------------=> UNTESTED (NOT recognizable)
LADD ------------------------=> UNTESTED (NOT recognizable)
FADD ------------------------=> UNTESTED (NOT recognizable)
DADD ------------------------=> INLINED and WORKING (RECognizable)
```

Figure 9: Final results (fragment) as displayed by SableVM after an execution of BTF test suite on PowerPC architecture.

```
UNTESTED - NOT COPYABLE           = 123
UNTESTED                          = 0
FAILING                           = 0
COPIED and WORKING (NOT FAILING)  = 209
UNDETERMINED                      = 0
TOTAL COPIED    (FAILING OR NOT)  = 209
RECOGNIZABLE   METHOD/BCODE NAME  = 228
NON-RECOGN. BY METHOD/BCODE NAME  = 104
```

Figure 10: Test results summary as displayed by SableVM after an execution of BTF test suite on HPPA architecture.

one hour. Most ports only took a few hours of work. For Linux 2.4 and GCC 3.3 we ported SableVM code-copying engine to 6 architectures: Alpha, i386, IA64, PowerPC, Sparc, HPPA. Where making code-copying work proved challenging we still used BTF to ensure proper execution of direct-threaded engine on m68k and s390 architectures.

We attribute this success to the ease of use (see Figures 9 and 10) and the comprehensive nature of our approach. This clearly shows the advantage of using specialized tools like BTF and SableVM's Testing Mode to ease porting efforts and improve VM reliability in a variety of different environments.

7 Conclusions

Code-copying removes dispatch overhead and improves branch prediction delivering a much better performance than the more standard direct-threading. Code-copying, however, requires ensuring that the copied code will actually execute properly in all situations. We developed a test suite with fine-grained, specialized tests written in Java bytecode assembly used by Jasmin. The suite tests all control flows visible in chunks of VM source code. In connection with special testing mode in SableVM it can be used to test code-copying engine and automatically report on findings regarding proper execution of bytecodes by this fast engine.

We developed a set of m4 macros to avoid cluttering the VM sources and separated the VM sources from the textual database containing information which bytecodes can be used for code-copying and which can not, depending on architecture, compiler and its version. This database is easy to maintain and update thus is an important improvement in practical usability of SableVM's code-copying engine.

Thanks to BTF testing SableVM's code-copying engine has been greatly simplified. It still requires manual execution and is an iterative (fixed-point) process, but, as we've seen in practice, with clear instructions even a person with no VM knowledge can successfully use it to port SableVM's code-copying engine to new architectures.

8 Future work

Code-copying presents itself an interesting alternative to standard, slower interpretation methods. Given its low implementation costs and exceptional performance the only issues that need to be addressed to enable it wider adoption are safety and maintenance in the presence of new compiler versions. We see that there are two main areas for improvement. One having to do with the VM itself (and its tools), and other being a new area of improvement within the static compiler, GCC.

To ensure full SableVM robustness we would need to further improve Bytecode Testing Framework to include tests for other, currently untested bytecodes. Enabling more bytecodes to be used for code-copying is expected to have a measurable positive influence on the VM performance by allowing for longer superinstructions which will eliminate even more dispatch overhead.

From our experiences with multiple architectures we also realized that due to compiler optimizations the correlation between the control flow paths in the source code and in the resulting binary is not full, therefore there are limits to what BTF can accomplish as a testing tool. Because of that we believe that in a long-term view a much better option would be to enhance a highly robust, optimizing static compiler like GCC with the support necessary for code-copying. Such support could ensure the proper ordering of basic blocks within code chunks or ensure the use of absolute addressing of jumps and calls where necessary. Our future work is therefore expected to be going towards improving the support for code-copying in highly optimizing compilers.

References

[1] David Bélanger. SableJIT: A retargetable just-in-time compiler. Master's thesis, McGill University, August 2004.

[2] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *SIGPLAN '03 Conference on Programming Language Design and Implementation*, 2003.

[3] M. Anton Ertl and David Gregg. Retargeting JIT compilers by using C-compiler generated executable code. In *Parallel Architecture and Compilation Techniques (PACT' 04)*, pages 41–50, 2004.

[4] M. Anton Ertl, Christian Thalinger, and Andreas Krall. Superinstructions and replication in the Cacao JVM interpreter. *Journal of .NET Technologies*, 4:25–32, 2006. Journal papers from *.NET Technologies 2006* conference.

[5] Etienne Gagnon and Laurie Hendren. SableVM: A research framework for the efficient execution of Java bytecode. In *Java Virtual Machine Research and Technology Symposium*, 2001.

[6] Etienne M. Gagnon. *A Portable Research Framework for the Execution of Java Bytecode*. PhD thesis, McGill University, 2002.

[7] Etienne M. Gagnon. Porting and tuning inline-threaded interpreters. In *CASCON 2003 workshop reports*, 2003.

[8] Jonathan Meyer and Daniel Reynaud. Jasmin - an assembler for the java virtual machine. `http://jasmin.sourceforge.net/`.

[9] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 291–300, New York, NY, USA, 1998. ACM Press.